DTU Library

Object oriented machine learning with a multicore real-time java processor: short paper

Pedersen, Rasmus Ulslev; Schoeberl, Martin

Published in:
Proceedings of the 8th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2010)

Link to article, DOI:
10.1145/1850771.1850782

Publication date:
2010

Document Version
Early version, also known as pre-print

Short Paper: Object Oriented Machine Learning with a Multicore Real-Time Java Processor

Rasmus Ulslev Pedersen
Embedded Software Laboratory
Department of Informatics
Copenhagen Business School
2000 Frederiksberg, Denmark
rup.inf@cbs.dk

Martin Schoeberl
Department of Informatics and
Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

ABSTRACT

The term intelligent systems is spreading beyond the data mining and machine learning communities. This presents new challenges that are fundamental to classical problems within object oriented programming and analysis. In this paper we investigate the use of a popular intelligent algorithm on a Java-based processor. The processor is a real-time enabled processor implemented on an FPGA, and we deploy a support vector machine on this processor. Furthermore, we show how this support vector machine can work on the Java-processor's multiple cores. This is a first step toward understanding how intelligent algorithms can be implemented on object-oriented Java systems with multiple cores in a hard real-time environment. Our experiments show significant speedup of the selected machine learning algorithm, and this can potentially be useful for other intelligent algorithms also.

1. INTRODUCTION

We explore machine learning on a multi-core version of the Java Optimized Processor (JOP) [6]. A field-programmable gate array (FPGA) can host up to 12 JOP cores. On each of those cores, we can install a machine learning algorithm. The intention is to speedup the machine learning and classification by parallelization on the multi-core system. The synchronization of the critical collection of data is a negligible part of the overall computational process.

Support vector machines (SVM) are part of a family of flexible machine learning algorithms for predicting structured objects and they are finding their way into mainstream computer science [2] and into embedded systems [3]. We use a standard dataset called Weather[1] to demonstrate the multicore-enabled SVM.

This paper describes a framework for using the SVM in a distributed environment with an emphasis on constrained computing [4].

[1] http://www.cs.waikato.ac.nz/~ml/weka/index.html

JTRES'10 August 19–21, 2010 Prague, Czech Republic


```
/**
 * Method getKernelOutput, which returns the kernel of two points.
 *
 * @param i1 – index of alpha_fp 1
 * @param i2 – index of alpha_fp 2
 * @return kernel output
 */
float getKernelOutputFloat(int i1, int i2) {

    kernelCalls++;

    return KFloat.kernel(i1, i2);
}
```

Figure 1: Normal Java code for the getKernelOutputFloat

Statistical learning theory has had a profound impact on learning theory over the last two decades, which is supported by the over 700 references at the SVM related site www.kernel-machines.org. Statistical learning theory has been developed and synthesized primarily by Vapnik [8]. He led the work on the support vector algorithm upon which this theory is based.

The paper is structured as follows. An overview of Support Vector Machines is presented in Section 2. We use the Java optimized processor (JOP) as the implementation platform, and this platform is introduced in Section 3. Experimental results are presented in Section 4, and the paper is concluded in Section 5.

2. SUPPORT VECTOR MACHINES

The SVM is an algorithm that is based mainly on work performed by Vladimir N. Vapnik and coworkers. It was presented in 1992 and has been the subject of much research since. We look at the algorithm from an application viewpoint and review its characteristics. A secondary purpose of this review is to introduce the definitions that play a central part in the following sections.

The SVM algorithm is a maximal margin algorithm. It seeks to place a hyperplane between classes of points such that the distance between the closest points are maximized. It is equivalent to maximum separation of the distance between the convex hulls enclosing the class member points. Vladimir Vapnik is respected as the researcher who primarily laid the groundwork for the support vector algorithm.

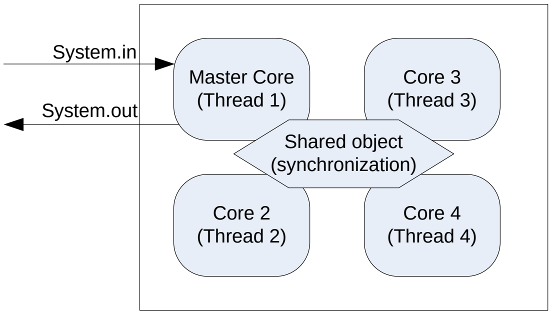

Figure 2: Overview of JOP CMP example setup for 4 cores

The kernel function k is implemented in Java. Part of the Java implementation is shown in Figure 1. The kernel function takes two inputs and delegates the computation to a static method kernel in class KFloat.

The functional output of the SVM formula is ± 1, which works as a classification or categorization of the unknown datum $\boldsymbol{x}$ into either the $+$ or $-$ class. An SVM model is constructed by summing a linear combination of training data (historical data) in feature space. Feature space is implicitly constructed by the use of kernels, k. A kernel is a dot product, also known as an inner product, in a space that is usually not of the same dimensionality as the original input space unless the kernel is just the standard inner product $\langle\ ,\ \rangle$.

The optimization problem is constructed by enforcing $|f(\boldsymbol{x}, \boldsymbol{\alpha}, b)| = 1$ for the support vectors. Support vectors (SV), are those data, $\boldsymbol{x}_i$, which have active constraints, $\alpha_i > 0$. If the data is not separable by a linear hyperplane in a kernel induced feature space, then it would not be possible to solve the problem if there was not an upper limit to the values of the active Lagrange multipliers. Consequently, the constraint, C, ensures that the optimization problem remains solvable.

3. THE JAVA OPTIMIZED PROCESSOR

The Java processor JOP [6] is a hardware implementation of the Java virtual machine (JVM). JOP is open source under the GNU GPL. Thus, it is freely available for research and educational purposes. JOP can be used as a research platform for low-level hardware development, system level research within the implementation of the JVM, and real-time application development for embedded Java. The most popular platform for JOP is the Cyclone FPGA from Altera. It is available from `www.jopdesign.com`.

We present our approach to embedded machine learning on a chip-multiprocessor version [5] of the JOP [6]. JOP is implemented in a field-programmable gate array (FPGA). The maximum number of processing cores of the chip-multicore (CMP) system depends on the size of the FPGA. Up to 8 cores fit into a medium sized low-cost FPGA (e.g., Altera Cyclon-II EP2C35), as found on the Altera DE2 board.

In Figure 2, we can see how one core supports the development process by providing System.in and System.out streams. That communication responsibility is covered by Thread 1.

To simplify the parallelization of the SVM algorithm we have built a small *executor* framework. The client of the framework has to create an executor with the problem size n (independent units of work) and provide a class that implements an interface with a single method execute(int nr). This method implements the unit of work and is automatically invoked by the framework n times. The framework automatically distributes the workload to all available processor cores. Therefore, several iterations of execute(int nr) execute in parallel. Any access to shared data needs to be properly synchronized.

Conceptually there is a single thread per core to execute the workload. The executor framework is more lightweight than starting a thread per unit of work. Our approach is similar, but simpler, than the *Executor* framework introduced in Java 1.5. The following listing shows the usage of our executor framework.

```
public static void main(String[] args) {
        Execute e = new Test();
        ParallelExecutor pe = new ParallelExecutor();
        pe.executeParallel(e, Test.N);
        Test.result();
}

private static class Test implements Execute {
        final static int N = 100;
        static int a[] = new int[N];

    // the work method for one iteration
        public void execute(int nr) {
                a[nr] = nr;
        }

        public static void result() {
                for (int i=0; i<N; ++i) {
                        System.out.println(a[i]);
                }
        }
}
```

The parallelization takes place in the $\sum_{i=1}^{l}$ part of the SVM formula. The parallel execution framework calculates the $\alpha_i y_i k(\boldsymbol{x}_i, \boldsymbol{x})$ part for each support vector. After the parallel part, the bias, b, is subtracted to reach the decision if the point belongs to the negative instances or the positive instances.

Algorithm 1 Classify point in parallel

Require: Trained SVM
Ensure: $y = x^n$
  $f_{svm} \Leftarrow 0$
  for all support vectors ($\alpha_i > 0$) **do**
    $f_{svm} += y$ {executed in parallel}
  end for
  Consolidate

From Algorithm 1 it is evident that some first level optimization is possible for the condition $\alpha_i > 0$. However, since we do not know in advance how many support vectors we have, this number is predetermined by the algorithmic designer. It is not difficult to obtain, as it can be found during the initialization phase for example.

4. EXPERIMENTS

In this section we demonstrate the worst case execution time analysis of the SVM on JOP. The control flow graph is created using the worst case execution time analysis tool called WCA [7, 1].

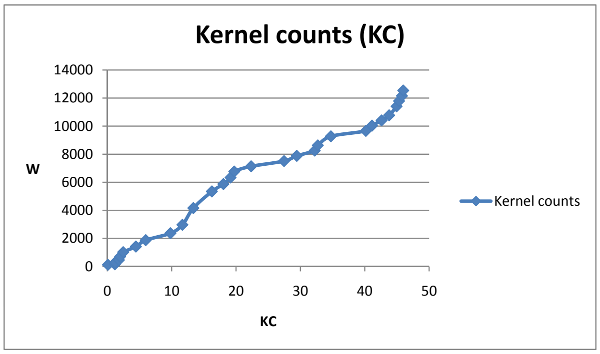

Figure 3: Optimization versus kernel counts

Cores	Classification
1	1,216,203
2	665,340
3	550,155

Table 1: Execution time in clock cycles for different number of cores on Weather data

The kernel code used to execute this example is introduced in Section 2. We can see how the kernel method in KFloat is converted to a control flow graph with basic blocks (code with no branches) and interconnecting vertices depicting the branches. Additional blocks are introduced for method invokations.

The SVM algorithm trains to optimize the objective function. The kernel[2] is a heavily accessed code section in any SVM. We have plotted the kernel counts versus the optimization function for the `Weather` data in Figure 3.

Using the JOP programs allow us to test the speed gains of using multiple processor cores. The experiments have been run in an FPGA platform with 1, 2, and 3 cores and a time-predictable memory arbitration scheme between the cores.

The speed gains on several cores, as shown in Table 1, are considerable. Doubling the number of cores gives almost a linear speedup. With three cores the bandwidth to main memory limits the additional speedup. This is an indication that more work on core local caches needs to be done on JOP. In general it is a sign that the communication overhead of running the multicore analysis is insignificant compared to the reduction in cycles of spreading the work across several processor cores. Figure 4 shows the speedup graphically.

5. CONCLUSION

In this paper we have taken the support vector machine algorithm, and demonstrated how it can be used in a Java multicore environment. The Java processor JOP is used as the execution platform, and the code is analyzed for the worst-case execution time. This is done for both a single CPU and multiple cores. The algorithm is parallelized on its most computational intensive points. We achieved linear scalability for two cores, and presented for the first time one of the most popular machine learning algorithms enabled for a real-time multicore execution environment. It is our conclusion that objected oriented intelligent algorithms are a subject of significant interest as we have demonstrated on popular family of algorithms on the JOP CMP processor.

[2] a method called getKernelOutputFloat

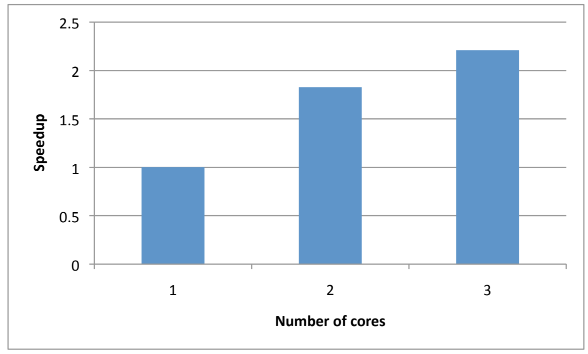

Figure 4: Speedup on Weather data with different core configurations

6. REFERENCES

[1] B. Huber. Worst-case execution time analysis for real-time Java. Master's thesis, Vienna University of Technology, Austria, 2009.

[2] T. Joachims, T. Hofmann, Y. Yue, and C.-N. Yu. Predicting structured objects with support vector machines. *Commun. ACM*, 52(11):97–104, 2009.

[3] R. Pedersen and M. Schoeberl. An embedded support vector machine. In *Proceedings of the Fourth Workshop on Intelligent Solutions in Embedded Systems (WISES 2006)*, pages 79–89, Vienna, Austria, June 2006.

[4] R. U. Pedersen. *Using Support Vector Machines for Distributed Machine Learning*. PhD thesis, Dept. of Computer Science, University of Copenhagen, 2005.

[5] C. Pitter and M. Schoeberl. A real-time Java chip-multiprocessor. *Trans. on Embedded Computing Sys.*, accepted for publication, 2010.

[6] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[7] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[8] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, NY, 1995.